Massachusetts Institute of Technology
Lincoln Laboratory

A Study of Gaps in Defensive Countermeasures for Web Security

Kevin Bauer
Thomas Hobson
Hamed Okhravi
Shannon Roberts
William Streilein

Technical Report 1196

October 18, 2015

This work is sponsored by the NSA's Cyber Defense Research & Technology Team under Air Force Contract #FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government. Distribution A: Public Release

Lexington Massachusetts

TABLE OF CONTENTS

			Page
	List of Figures		v
	List of Tables		vii
1.	EXECUTIVE SUMMARY		1
2.	INTRODUCTION		3
	2.1	Goals	3
	2.2	Scope	3
3.	METHODOLOGY		5
	3.1	Categorization of Web-based Attacks	5
	3.2	Attack Prioritization Process	5
	3.3	Gap Discovery Process	7
	3.4	Gap Selection Process	7
	3.5	Gap Treatment and Classification Process	7
4.	GAPS AND RESEARCH DIRECTIONS		11
	4.1	Lack of effective defenses to combat Cross-Site Scripting (XSS)	11
	4.2	Lack of effective defenses to combat Code Injection and Code Reuse	15
	4.3	Lack of effective defenses to combat Session Fixation	18
	4.4	Lack of effective defenses to combat Privilege Escalation	20
	4.5	Lack of effective defenses to combat Logic Vulnerabilities	23
	4.6	Lack of effective defenses to combat File Inclusion	25
	4.7	Lack of effective defenses to combat Drive-by Downloads	27
	4.8	Lack of effective defenses to combat Clickjacking	29
	4.9	Lack of effective defenses to combat Plug-in Attacks	31
5.	SUMMARY AND CONCLUSION		33
Appendix A: Summary of Deployment Gaps			35
	References		37

LIST OF FIGURES

Figure
No. **Page**

This page intentionally left blank.

LIST OF TABLES

Table No.		Page
1	Prioritization of attack areas	6
A.1	A summary of web-based countermeasures and their deployment gaps.	35

1. EXECUTIVE SUMMARY

Web-based attacks are a prominent class of cyber attacks in today's networks. They are attacks that violate the security properties of web servers, web applications, web portals, web browsers, and web services. They can damage confidentiality, integrity, and availability of systems and networks and pose a significant threat to both systems connected to open, public networks (i.e. the Internet) and those that reside on closed, private networks. In their impact and sophistication, web-based attacks are on par with host-based attacks. Most web-based attacks are a form of the *confused deputy problem* in which one party is fooled about the identity or authority of another party. Virtually all web-based attacks are also a form of *input validation problem* where the target fails to properly check a potentially malicious, user provided input.

Traditionally, simple defenses against web-based attacks, such as input sanitization, provide little protection against a motivated attacker with simple evasion capabilities and often have impractically high false positive and false negative rates. More effective defenses in this domain often either require significant modifications to servers and infrastructures, thus violating the federated model of such networks, or they impose high computational or operator overheads. As a result, the domain of web-based attacks requires significant research and development efforts to provide practical, effective defenses.

In this report, we highlight some of the most important deployment challenges and gaps related to web-based defenses, which can be used to guide future research and development in this area.

This page intentionally left blank.

2. INTRODUCTION

In this report, we analyze and enumerate the gaps that exist in the area of countermeasure applications. Countermeasure applications refer to conducting and applying research to prevent adversarial activity on US Government (USG) networks through the application of defensive techniques. Cyber Network Defense (CND) activities often include actions such as *deterrence*, *detection*, *protection*, and *reaction* to attacks.

2.1 GOALS

The main focus of this report is on *protection*, i.e., stopping or mitigating the impact of cyber attacks by applying defensive techniques prior to their occurrence. Of particular interest are defensive techniques that can mitigate attacks that use known and unknown vulnerabilities of known types (i.e. zero-day attacks). Attacks of unknown types are out of the scope of this analysis due to the inherent difficulty to reason about them.

2.2 SCOPE

In cyber security, some domains are better understood than others. For example, while the domain of memory corruption attacks is well studied with a rigorous taxonomy [41], the domain of web-based attacks is understudied. As a result, for this report, we limit the scope of our study to web-based attacks.

Web-based attacks are those that violate the security policies of websites, web applications, and web services. We simply refer to all of the web-based components and systems as web services. W3C defines a web service as “a software system designed to support interoperable machine-to-machine interaction over a network” [44]. Web services may be accessible over a public network (e.g. DISA’s webpage) or they may be isolated in a closed, private enclave (e.g. an information portal accessible over SIPRNet). Web services often use well-known coding languages (e.g. PHP, JavaScript, SQL, and Ruby), markup languages (e.g. XML, HTML, XACML, WSDL, and SAML), and protocols (e.g. HTTP, HTTPS, and SOAP). Web services often have a client-server model in which the content is mainly provide by the server, but the peer-to-peer model is also used in some web services.

This page intentionally left blank.

3. METHODOLOGY

3.1 CATEGORIZATION OF WEB-BASED ATTACKS

Web-based attacks are those that violate the confidentiality, integrity, and/or availability of web services. Although similar in their concepts and impact, web-based attacks are distinguished from other domains of cyber attack such as memory corruption, by their abstraction layer. Web-based attacks work at the abstraction layer of web applications, web protocols, and web languages.

At the highest level of abstraction, most web-based attacks, with the exception of some forms of drive-by downloads [26], are a form of the *confused deputy problem.* The confused deputy problem refers to attacks in which one party is fooled about the identity or authority of another party. To the best of our knowledge, all web-based attacks are also an *input validation problem* where user provided input is not checked or sanitized properly.

There are two major classes of defenses against web-based attacks: exploit defenses and payload defenses. Exploit defenses try to stop the exploitation of a web-based vulnerability. For example, character escaping techniques [15] to mitigate SQL injection attacks are a form of exploit defense because, if done properly, they can stop exploitation of the SQL injection vulnerability. On the other hand, payload defenses, try to mitigate the impact of the attack after the initial exploitation. For example, iFrame isolation techniques [7] assume that cross-site scripting (XSS) exploits can happen, so they try to limit the compromise to a single iFrame.

Categorizing web-based defenses beyond these major classes is challenging. Categorization based on the mechanism can be misleading. For example, input sanitization is used in numerous defenses to mitigate different types of web-based attacks, but depending on the type of attacks mitigated, those defenses can have different properties. For instance, an input sanitization defense applied to a web server to mitigate session fixation attacks can have a very low false positive, whereas input sanitization applied to the web browser to mitigate client-side XSS attacks may have a very high false positive rate. Categorization based on the side of defense (client-side vs. server-side) can also be misleading. For example, different server-side defenses may protect against incongruent types of web-based attacks with varying properties.

One type of categorization that can form a more rational basis for comparison of web-based defenses is based on the attack types mitigated. There are relatively few types of web-based attacks, which make this categorization simpler. In addition, defenses mitigating the same attack have the same ultimate goal, which allows a better comparison of their properties. As a result, in this report, we use an attack-based categorization to study the gaps in web-based defenses. The attack types studied in this report are: Cross-Site Scripting (XSS), Code Injection and Code Reuse, Drive-by Downloads, Clickjacking, Logic Vulnerabilities, Session Fixation, File Inclusion, Privilege Escalation, and Plugin Attacks.

3.2 ATTACK PRIORITIZATION PROCESS

One possibility for prioritizing research efforts is to look at the risk posed by each attack area. For a quantitative computation of risk we need to understand the likelihood and impact of attacks

TABLE 1

Prioritization of attack areas

	Cenzic	WHID	OWASP	Merged Priority
XSS	1	2	3	High
Code Injection and Reuse	4	1	1	High
Session fixation	3	3	2	High
Privilege escalation	2	6	7	Med
Logic vulnerabilities	2	4	2,5,7	Med
File inclusion	N/A	5	4	Med
Drive-by downloads	N/A	N/A	10	Low
Clickjacking	N/A	7	N/A	Low
Plug-in attacks	N/A	N/A	9	Low

occurring in each area. Reliable and well-classified data sources are lacking for the likelihood component and even more so for the impact component, which is somewhat unique to each individual organization. Most data sources contain high levels of subjectivity in how attacks are classified and often report attacks by types that are overlapping or are overly abstract. This makes them challenging and often misleading to map to particular use cases such as the attack areas described above. In an effort to minimize the deficiencies inherent in any individual data source we have examined multiple recent web attack sources, mapped them to our attack areas, and averaged their prioritizations to come up with an overall prioritization level.

We examined three sources: the annual Cenzic vulnerability trends report [8], the Web Hacking Incident Database (WHID) [39], and the OWASP Top 10 Ranking [28]. The Cenzic report provides some insight into the likelihood of attack by providing the number of vulnerabilities per attack type as reported via a web application vulnerability detection product. The WHID reports incidents of targeted attacks that have been publicly reported. As this database is specifically filtered to include targeted attacks it may provide some information not only about likelihood but also about impact given that targeted attacks may be more relevant and impactful for the organizations under consideration. Finally, the OWASP ranking provides a consensus of the top 10 attacks as reached by experts in the web security community. Being derived from the experience of subject matter experts, it incorporates subjective information about both the likelihood and impact.

While not strictly agreeing upon an ordering, the sources were generally in agreement about whether the attacks were of high, medium, or low priority. Table 1 lists these priorities by attack type and shows the relative ranking for each data source. Some data sources did not classify attack types in a way that mapped cleanly to the attack areas described above, as indicated by N/A in the table.

3.3 GAP DISCOVERY PROCESS

For the purposes of this study, we focus on the open literature and publicly known defensive techniques. We reviewed papers and studies, mainly from the prominent security conferences such as the IEEE Symposium on Security and Privacy (Oakland), Network and Distributed Systems Security (NDSS), ACM Computers and Communications Security (CCS), USENIX Security Symposium, Annual Computer Security Applications Conference (ACSAC), and the Symposium on Research in Attacks, Intrusions, and Defenses (RAID).

We initially surveyed the titles and abstracts for the past four years of each conferences proceedings. We then down-selected this very large list of papers based on their relevance to attack analysis. We read the papers from this refined list and extracted any relevant gaps based upon our own experience and the context of the paper and the state research contribution.

We have also incorporated publication, reports, and information from other sources such as white papers and threat reports, as necessary.

3.4 GAP SELECTION PROCESS

The gap selection process was performed during team meetings and began with individual team members independently determining if a gap was worthy of carrying forward. A majority vote was then used to determine whether to keep it as a gap or not.

3.5 GAP TREATMENT AND CLASSIFICATION PROCESS

3.5.1 Types of Gaps

There are three major types of gaps in existing cyber defense:

- Technology Gap: an effective defense does not yet exist against a type of attack.
- Deployment Gap: defenses have been proposed or implemented against an attack type (open source or commercial), but all of the existing defenses have impracticalities that impede their adoption. For example, if all known defenses against an attack type have unacceptably high overhead or false positive rates, a deployment gap exists for such defenses.
- Practice Gap: practical and effective defenses exist in the community, yet their are not adopted widely, perhaps because of the lack of awareness or incentives.

In this report, we primarily focus on deployment gaps. Note, however, that the distinction between these areas are not well established and there are many gray areass. For example, the amount of acceptable overhead varies from system to system, which means that a deployment gap in one system can be interpreted as a practice gap in another. Moreover, each deployment gap eventually points to a technology gap. For example, knowing that all existing defenses against an attack have high overhead (a deployment gap) points to the fact that faster defenses have to be developed against that attack type (a technology gap)

3.5.2 Deployment Challenge Categorization

Deployment challenges are weaknesses or impracticalities in defensive techniques that impede or disincentivize their usage. Deployment challenges of web-based defenses can be categorized into three major classes: challenges related to protection, overhead, and compatibility.

A. Protection

1. **Effectiveness**: captures the strength of the security policy enforced by a defensive technique. If a defense can easily be evaded by adapting the attack without violating the security policy enforced by the defense, the defense has low effectiveness. Otherwise, if most malicious attacks necessarily violate the security policy (and therefore are stopped by the defense), the defense has high effectiveness.
2. **False Positive Rate**: is the rate at which benign operations are incorrectly flagged as violations by the defense. A low false positive rate is very important for practical adoption of a defense. Defenses with high false positive rates can create the habit of ignoring alarms for the user which, in turn, negatively impacts the security of the system in cases where attacks are correctly detected.
3. **False Negative Rate**: is the rate at which attacks are incorrectly presumed benign by the defense. High false negative rates are an indication of a weak defense. Note that effectiveness mainly captures the resilience of a defense against evasion attempts, whereas the false negative rate is the amount of intrinsically missed attacks. In both cases, the security policy enforced by the defense is not violated.

B. Overhead

1. **Performance**: is the defense's effect on speed as caused by processing or performing a task. Low performance overhead is crucial in widespread adoption of defenses. Some studies argue that an overhead of higher than 10% is considered unacceptable for practical adoption of defenses [41].
2. **Communication**: is the additional communication that is incurred by the defense.
3. **Human**: is the amount of manual work necessary to operate the defense.

C. Compatibility

1. **Source**: is the defense's ability to be applied to source code without manual modification.
2. **Binary**: is the defense's ability to be applied to binaries without manual modification. Note that techniques that require access to the source code are not binary compatible.
3. **Infrastructure**: is the defense's ability to be applied to existing infrastructure such as networking appliances, servers, and network architectures, without manual modification. For example, a technique that runs an extra piece of code on the web server to secure it against injection attacks is infrastructure compatible, while another one that requires modifications to the enterprise gateway is not.

4. **Federation**: is the defense's ability to be applied to a federation without controlling every entity. The Internet, the World Wide Web, and even some closed networks consist of independently owned and managed entities. In such an environment, it is important that a defense can be applied effectively to a subset of the entities under defender's control. A defense that can be applied to one side of the web services (often the client) supports federation. Otherwise, if the defense has to be applied to both clients and servers for it to be effective, it does not support federation because it requires control over both entities.
5. **Modularity**: is the defense's ability to be applied to each module independently. In other words, if the defense must be applied to every component of the system for it to work properly, it is not modular. Otherwise, if it can be applied to some components, but not others and still works properly, it is modular. An example of a modular technique is a static code analysis technique that finds vulnerabilities in the code, because some modules can be analyzed without the necessity to analyze all of them. An example of a non-modular technique is a data tagging mechanism to implement taint tracking in the browser, because all browser modules must be augmented with the mechanism; otherwise, the browser will crash. For instance, if the HTML parsing module is augmented with tagging, but the JavaScript module is not, the browser will crash.

We refer to these deployment challenges in the rest of the report using their numbers, e.g., A.2 refers to the false positive rate and C.4 refers to the federation support.

This page intentionally left blank.

4. GAPS AND RESEARCH DIRECTIONS

4.1 LACK OF EFFECTIVE DEFENSES TO COMBAT CROSS-SITE SCRIPTING (XSS)

4.1.1 Attack Description

Cross-site scripting (XSS) vulnerabilities frequently top the list of web-based attacks [8]. XSS vulnerabilities can occur whenever a web page allows for user input. In its simplest form, a XSS vulnerability occurs when script is injected into a webpage for which it was not intended [35, 43]. Most, but not all, XSS vulnerabilities involve injection of JavaScript into webpages. According to the Same Origin Policy, since the script comes from a trusted web application, the browser processes the request, returns a response, and most importantly, does not flag an error. The consequences of XSS vulnerabilities are diverse, but often lead to stolen credentials, unwarranted privileges, and impersonation attempts. Related to XSS are other vulnerabilities such as cross-channel scripting (XCS) and cross-site request forgery (CSRF). XCS is a special from of XSS where a channel, e.g., FTP, is used to inject an XSS exploit [6]. CSRF is a near opposite of XSS in that a malicious site sends a request to an honest site via a trusted user; in this way, the attacker is now taking advantage of the trust that the site has in the user [3]. Though these attacks are distinct from XSS, they can be treated in a similar manner with respect to their prevention.

XSS attacks fall into two categories based upon the source of their incorrect code: server-side (i.e., non-persistent (reflected) and persistent (stored) XSS vulnerabilities) and client-side (i.e., DOM-based, plug-in, and content sniffing XSS vulnerabilities) [7]. Once the script is maliciously crafted and goes to the server, if the request is stored in the servers archive, the XSS vulnerability is known as persistent (stored). If the server responds with the malicious code copied in the response, the XSS vulnerability is known as non-persistent (reflected) [12]. DOM-based XSS vulnerabilities occur when script is injected into a DOM operation, such as *document.write*. Plug-in XSS vulnerabilities occur when malicious script is inserted by plug-ins while content sniffing XSS vulnerabilities occur when a file, e.g., an image or PDF file, is interpreted as a script file, with this script file containing malicious code [7].

Common mitigations of XSS vulnerabilities include input validation/sanitization, regular expressions, character escaping, and client-side script disabling. Input validation or sanitization techniques attempt to sanitize untrusted code that may contain malicious script. Regular expressions are used to identify and remove malicious content from outgoing requests [24, 40]. Character escaping literally escapes special characters within the user input, thereby disallowing any code execution. Lastly, script disabling is when the user disables scripts from running within their web browser. These four common techniques are relatively simple to deploy and, at an intuitive level, provide protection against XSS vulnerabilities. At the same time, they have their disadvantages. For example, most websites require JavaScript and as such, script disabling often results in reduced functionality of these websites. The use of character escaping and regular expressions results in many false positives and false negatives (A.2, A.3). Sanitization is subject to the quality of its implementation as developers may apply sanitizers in the wrong order, in the wrong context, or simply not frequently enough [35]. More importantly, none of these traditional defenses are bypassable, i.e., attacker's can easily evade them (A.1).

Outside of these mitigation techniques is Content Security Policy (CSP), a protocol designed to prevent XSS vulnerabilities by essentially creating a white list of sources that a web page should trust [12]. CSP enforces an access control policy on different constituents of a webpage. CSP differs from the techniques that were presented previously as it is a protocol that must be followed, not a tool or technique that is applied to a web browser, server, etc. Many modern web browsers implement their own version of Content Security Policy, but CSP is only deployed by a small fraction of websites [46].

4.1.2 Defense: Chrome's XSS Auditor

Description Of Defense
XSS Auditor is similar to regular expression defenses in that it uses regular expressions to identify malicious content. It defers from past approaches since it detects the malicious content after the HTML is parsed and before it is passed to the JavaScript engine. In this way, content that only becomes malicious after being parsed by the HTML parser can be caught before being sent to the JavaScript interpreter where it is actually executed. If XSS Auditor detects injected script, it replaces it with a benign value or removes it entirely. XSS Auditor is implemented in Google Chrome [24].

Description of Deployment Gaps
Though promising, the XSS Auditor does not prevent XSS Vulnerabilities that do not pass from the HTML parser to the JavaScript Interpreter, thereby limiting its scope and allowing attackers an easy bypass route (A.1). As the XSS Auditor verifies injected code by confirming that it is not in the request using string matching, if attackers craft the attack such that the request and the injected code match, they can once again, bypass the XSS Auditor. This leads to a high false positive and false negative rate (A.2, A.3). Lastly, XSS Auditor is only available in Google Chrome (C.2) [24].

4.1.3 Defense: Script Separation

Description Of Defense
With respect to preventing server-side XSS vulnerabilities, especially for legacy web applications, one solution proposes to separate code and data such that all inline JavaScript is contained in external files. Based on CSP, only JavaScript contained in these external files is executed [12].

Description of Deployment Gaps
As this solution only prevents server-side XSS vulnerabilities when used in combination with other techniques, i.e., CSP, its effectiveness is limited (A.1). On top of that, this solution only identifies 50-70% of inline JavaScript (A.1). This solution has only been tested on open source web applications (A.1) and requires binary code to be rewritten on the server (C.1, C.4). Lastly, overhead associated with this application is unclear (B.1) [12].

4.1.4 Defense: Training-Based Countermeasure

Description Of Defense
Using a two-step process (training and runtime auto-correction), ScriptGard aims to prevent XSS and XCS vulnerabilities. During the training process, positive taint tracking is used to develop a sanitization cache to map paths and the correct use of sanitizers. During the runtime auto-correction phase, the sanitization cache is used to ensure the correct ordering of sanitization [35].

Description of Deployment Gaps
This solution only repairs sanitizers; it is not a complete mitigation tool, i.e., if the correct sanitizers are not implemented or used to begin with, this solution is of no use (A.1). In addition, this solution has only been implemented in Internet Explorer (C.2). Though the repair only occurs in the second phase (runtime auto-correction), the training phase can sometimes result in overheads of 175 times, thereby severly limiting this solution's deployment (B.1). Lastly, developers are needed to indicate the context in which sanitizers should be used, causing human overhead (B.3) [35].

4.1.5 Defense: Taint Tracking

Description Of Defense
Another solution to preventing XSS vulnerabilities includes tracking the flow of attacker-controlled data and using HTML and JavaScript parsers to detect tainted data. Such a solution has low false positive and false negative rates [40]. A similar solution to the prevention of DOM-based XSS vulnerabilities relies on taint tracking in the JavaScript and DOM engines. After a tainted string is passed to a sink, a report is generated and shown to the user [23].

Description of Deployment Gaps
Both of these taint tracking solutions only prevent DOM-based XSS vulnerabilities (A.1), and they have only been implemented in open-source browsers (C.2). Extending it to other browsers would require binary changes. Though these solutions have been tested with known DOM-based XSS vulnerabilities, it is not known how well they perform against XSS attacks that are crafted to evade these solutions (A.1) [23, 40]. In addition, with respect to the taint tracking solution that is implemented within the JavaScript and DOM engines, features designed to optimize the web browser's performance must be disabled, thereby indicating that this solution is not modular as other modules must be modified in order for the solution to properly function (C.5) Lastly, the overhead associated with this solution is unclear (B.1) [23].

4.1.6 Defense: iFrame Isolation

Description Of Defense
Taken a different approach, one solution to prevent XSS vulnerabilities is to separate webpages into iFrames such that communication between different web page sources is monitored. Monitoring whether one iFrame tries to change the content of another iFrame limits XSS worm propagation [7].

Description of Deployment Gaps
Though promising, this technique of iFrame isolation only prevents the spread of XSS vulnerabilities, not damage associated with the XSS vulnerability (A.1). It also has only been developed for

social networking sites. Outside of this mitigations effectiveness issues, it requires changes within the browser, server, and infrastructure (C.1, C.2, C.3, C,4) [7].

4.1.7 Defense: Mutation-Based XSS (mXSS) Countermeasure

Description Of Defense

For a specific type of web-based attack, known as a mutation-based XSS (mXSS) vulnerability, two solutions that utilize relatively simple techniques are identified. The first server-side solution disallows special characters, appends a trailing whitespace to text, and utilizes percent encoding for special characters. The second client-side solution overwrites access of particular commands, i.e. *innerHTML* and *outerHTML*, such that they are treated and handled as XML [16].

Description of Deployment Gaps

The server-side solution requires code to be modified on the server (C.2, C.4). Though the client-side solution does not require code modification, the end user may have to apply it, thereby introducing much variability, and possibly overhead (B.3). The overhead associated with the server-side solution is unclear (B.1). The client-side solution results in nearly a 30% increase for page-load times and a 12% increase for user perceived page-load times (B.1) [16].

4.1.8 Research Direction

Many of the defenses designed to combat XSS suffer from problems with effectiveness (A.1), performance overhead (B.1), and binary compatibility (C.2). Of particular importance is lack of effective defenses in this domain as most current defenses are easily bypassable. Future research in this area should focus on defensive techniques that are not easily evadable and that prevent all stages of an XSS attack. This can be done by focusing on fundamental properties of XSS attacks (i.e., the fact that server's identity is spoofed), rather than focusing on small artifacts of current attacks (e.g., usage of special characters in the malicious input) that can be evaded by a medium-low resourced attacker. Moreover, research should focus on designing more effective defenses for different types of XSS attacks (persistent, non-persistent, and DOM-based). It is also desired that future defenses for XSS attacks are binary compatible and exert little, if any, overhead.

4.2 LACK OF EFFECTIVE DEFENSES TO COMBAT CODE INJECTION AND CODE REUSE

4.2.1 Attack Description

Traditionally, *code injection* has been a method of exploiting memory corruption vulnerabilities (e.g., a buffer overflow) in binary executables and libraries. Exploits could be crafted by overwriting a function's return address or a code pointer with the memory location of an attacker-supplied payload containing executable code (e.g., shellcode) residing in a buffer on the stack. However, with the adoption of a non-executable stack (NX bit or Data Execution Prevention), such code injection attacks have been rendered ineffective, as this defense eliminates the ability to execute attacker-supplied code on the stack. *Code reuse* attacks can defeat these defenses by constructing attack payloads that combine code fragments that already exist within the binary's code segment. Examples of traditional code reuse attacks include *return-to-libc* [36], *return-oriented programming* [32], and *jump-oriented programming* [5].

Web applications can also suffer from memory corruption vulnerabilities that enable code injection and/or code reuse. However, the attack surface of web applications is much richer than traditional binaries, as web applications are often written in one or more high-level interpreted languages such as PHP, Perl, and/or SQL. While memory corruption bugs could exist within the interpreter of these languages, an often easier route of attack is to directly inject code written in the high-level language itself, often through an improperly validated input channel. For example, an HTTP request parameter whose value is interpreted as part of a PHP statement may be vulnerable to injection of arbitrary PHP code. Similarly, input parameters that are used to construct an SQL statement may be vulnerable to SQL injection, where attacker-supplied inputs are passed directly into an SQL statement without any input validation to ensure that the input contains data, not code. Code injection attacks against web applications and their back-end databases have the potential to impact the confidentiality, integrity, and/or availability of the web service.

More recently, code reuse attacks have been demonstrated within the context of PHP web applications [13]. Code reuse attacks against PHP applications may be possible if an application has an object injection vulnerability. The attacker injects an object into the web application through an input channel such as an HTTP request parameter. Prior to injection, the attacker can set the properties of the object in a manner to influence the data and control flow of the web application. As such, these attacks are often called *Property Oriented Programming* (POP) attacks, and like code injection attacks, can lead to the execution of arbitrary code within the web application.

Both code injection and code reuse attacks on web applications are made possible by improper input validation. One defense for these attacks is simply to validate all inputs using existing APIs that sanitize inputs. However, this general solution is expensive, as it requires the programmer to find all instances of the input validation vulnerability and apply source-level changes to validate the inputs. In addition, this solution may not be applicable to legacy web applications, for which there is limited developer support. Consequently, contemporary research efforts have endeavored to provide defenses against code injection/reuse attacks. We survey these defenses, along with their respective obstacles to deployment, in the sections below.

4.2.2 Defense: Static Analysis-Based Detection

Description Of Defense
One approach to the detection of code reuse attacks in PHP web applications is to use static analysis of the application's source code to identify PHP object injection vulnerabilities [11]. One challenge of this approach is that most static analysis frameworks for PHP do not support PHP's object-oriented features, which are of particular importance to attacks that exploit PHP object injection. This approach combines taint analysis, data flow analysis and an analysis of so-called "magic methods" that are related to object-oriented programming (e.g., constructors and destructors) to identify all PHP object injection vulnerabilities that may exist within a particular web application.

Another approach applies static analysis to detect "second-order" vulnerabilities in web applications [10]. Second-order vulnerabilities occur when a malicious value is stored in a persistent data store (e.g., in memory or a database) that can later be retrieved by that application. (This is in contrast to injections where the input is instantly used to read from a data store.)

Description of Deployment Gaps
In an evaluation of ten real-world PHP object injection vulnerabilities, the former approach found 30 new vulnerabilities, but also suffered from false positives and false negatives (A.2, A.3). More importantly, however, is the performance overhead: on average, the prototype required eight minutes and 2 GB of memory to perform the necessary analyses for a given application (B.1).

With regard to detecting second-order vulnerabilities, the later approach does not consider all potential persistent data stores; as such, it may not be effective if databases or the PHP `$_SESSION` array is not used. Additionally, the reported false positive rate is impractically high, 21% (A.2), and it is not compatible with existing PHP binaries (C.2).

4.2.3 Defense: Taint Tracking-Based Detection

Description Of Defense
To detect code injection within the context of server-side web applications that generate SQL and NoSQL queries, *Diglossia* distinguishes code from data in generated queries and determines which parts of the query are user-generated through efficient taint tracking [37]. The approach maps all application-generated characters to "shadow" characters that do not occur in the user input. Any non-shadow characters in the shadow query are thus tainted by user input. Diglossia parses the original query along with the shadow query and ensures that both are syntactically isomorphic and that all code within the shadow query is in the shadow characters, which means it was generated by the application, not the user input. A prototype is implemented as an extension to the PHP interpreter.

Description of Deployment Gaps
Diglossia is evaluated on a test suite of 11 PHP web applications. Reported performance overhead is negligible, code injection detection accuracy is perfect with no false positives or negatives reported, and no manual modification to PHP source code is required. However, the Diglossia PHP interpreter extension is necessary, which presents binary compatibility obstacles (C.2). Additionally, if the application developer intentionally includes user input as part of a SQL or NoSQL query, Diglossia

will report a code injection attack when the application runs (A.2). This is an important limitation, as many code injection vulnerabilities do, in fact, insert input user input directly into a query string (this is a bad programming practice, but nonetheless may occur in legacy web applications). As such, this approach may not be directly applicable to all PHP source code (C.1).

4.2.4 Research Direction

While the domain of code injection and code reuse attacks is well-studied for host-based attacks, it is much less mature for web-based attacks. Ad-hoc research has shown different possibilities for such attacks in modern systems (as discussed above), but the scope, classes, and scale of code injection/reuse vulnerabilities are yet unknown. Research should first focus on studying different classes, types, and possibilities of code injection/reuse attacks in web-based systems. Questions such as the following should be answered first: which components of a modern web-based system are usually vulnerable to such attacks? Is it only the scripting languages (e.g. PHP) or can they be applied to markup languages and protocols? What can an attacker do by exploiting each class? etc. Then, more effective defenses should be researched to combat such attacks. Since only small areas of web-based code injection/reuse attacks are studied, it is unknown to what extent the current defenses fall short of solving the entire problem. Having said that, both defenses reviewed in this report have issues with respect to high false positive rates (A.2) and binary compatibility (C.2). Future defenses to prevent against code injection and reuse should be applicable without changes to binary and should achieve much lower false positive rates to be widely applicable.

4.3 LACK OF EFFECTIVE DEFENSES TO COMBAT SESSION FIXATION

4.3.1 Attack Description

Session fixation attacks happen when an attacker can fake another user's session identifier (ID). Since session ID is often used by the web servers to allow users to have a persistent session and avoid having to enter their credential for every page separate, session fixation attacks allow an attacker to navigate to sensitive pages and access user's sensitive data without knowing user's credentials (e.g. username/password). Web servers that accept session IDs in the URL or POST data are especially vulnerable to session fixation attacks.

To understand how session fixation attacks work, consider the following example. Alice, an analyst, usually logs into an information portal `https://portal.com/` to query information she wants to analyze. She uses secure passwords to log into the portal. The portal tracks its user using a session ID that is sent back and forth in the URL. Malice wants to access Alice's private folder on the portal, so she crafts a link `https://portal.com/?SID=KNOWN_TO_MALICE` and sends it to Alice in a phishing email saying "See these reports on the portal". Alice clicks on the link and logs in using her credentials. From this point on, Malice can access Alice's private areas of the portal by navigating to `https://portal.com/?SID=KNOWN_TO_MALICE`. Because the portal keeps track of its user by the session ID, it is fooled to think Malice is Alice (i.e. confused deputy problem). There are also other variants of this attack with server generated session IDs.

Session fixation attacks can be detected by emulating a session hijacking attacker [42], in a process similar to fuzzing.

Session fixation attacks are simpler to mitigate than many other web-based attacks because only the session handling part of the server should be modified. In contrast, XSS attacks can happen potentially from any user provided input to the web service. Simple defenses such as changing the session ID every time the user logs in, tying session ID to stronger identifiers such as SSL session ID, and frequently changing the session ID can mitigate this attack to a large extent. As a result, less research has been done for defenses against session fixation compared to some other web-based attacks.

4.3.2 Defense: Code-Level Countermeasures

Description Of Defense
Session fixation attacks can be mitigated by analyzing the server source code [19]. Specifically, if in the server source code, the session ID is generated before the authentication, the session ID is not reissued after the authentication, and the code accepts the session ID from the user, it is vulnerable to session fixation attacks. A vulnerable server code can be fixed by breaking any of those conditions. For example, if the session ID is reissued after the authentication, it can mitigate the attack. A similar defense can be applied at the framework-level by analyzing the incoming HTTP requests and session ID values.

Description of Deployment Gaps
Code-level countermeasures unfortunately have various deployment gaps. First, they require access

to the source code of the server which makes them binary incompatible (C.2). Fully automated source code analyses can also be challenging, so in many cases such an analysis may require manual annotations (C.1). Moreover, code-level defenses also lack federation support because they require access to the web server (C.4).

4.3.3 Defense: Reverse-Proxy Countermeasures

Description Of Defense
A reverse-proxy defense works by assigning an extra proxy-generated session ID (PSID) to the server-generated session ID (SID) [19]. The PSID is refreshed after authentication which creates the illusion of SID re-issuance by the server. HTTP requests that do not carry the correct SID+PSID combination will be treated as a new request and will be assigned fresh PSIDs. As a results, session fixation attacks that spoof the SID are mitigated by the fresh PSID.

Description of Deployment Gaps
In contrast to code-level countermeasures, reverse-proxy defenses do not require access to the source code, or even the server itself. A reverse-proxy technique can be applied to the border gateway of an enterprise to protect against session fixation attacks targeting various external web servers. However, the reverse-proxy defense require modification to the infrastructure (C.3). Moreover, the reverse-proxy requires a centralized solution. In other words, all external web communications should go through the same proxy, and they cannot be load-balanced across multiple outgoing proxies, because one proxy can break another proxy's already established session. This indicates a scalability issue that may induce significant overhead when implemented (B.1).

4.3.4 Research Direction

Session fixation attacks are one of the easier research targets to solve in short term because they only require modification of the session management part of a web server. Research in this area should develop new session handling methods that are not accessible to user inputs. For example, many existing servers use user-provided session ID which creates attack opportunities. A challenge in session fixation attacks is the requirement to modify the server which creates compatibility problems. In fact, both defenses that address session fixation suffer from source (C.1) and binary compatibility (C.2) issues. Future defenses should address session fixation ideally without creating source/binary incompatibility.

4.4 LACK OF EFFECTIVE DEFENSES TO COMBAT PRIVILEGE ESCALATION

4.4.1 Attack Description

Privilege escalation attacks happen when users gains access to a web service beyond the content they are allowed to access. This happens, for example, by exploiting an implementation flaw in the web service code or logic. Privilege escalation can also happen because of missing or incorrect authorizations in the web servers [27]. Privilege escalation problems are exacerbated by three challenges: 1) web-services do not have native access control similar to operating systems, 2) the connection to the back-end database is often through a superuser (admin) which allows a potentially compromised front-end unlimited access to sensitive data, and 3) lack of a consistent framework makes authorization enforcement error-prone and incorrect in many cases.

Other classes of web-based attacks can also be considered a form of privilege escalation. For instance, XSS can also be considered a privilege escalation attack in the sense that the attackers gain access to parts of a page (iFrames) that they are not normally allowed to access. However, in this section, we limit the analysis to low-level code vulnerabilities that are not included in the other attack classes.

Defense against privilege escalation attacks often limit the impact of such attacks by creating isolation among the various web service components and limiting their interaction to the bare minimum. Many such defenses, in their core, are applying the *principle of least privileges* to the web service.

Privilege escalation defenses also provide the additional benefit of limiting the potential damages of other web-based attacks. For example, iFrame isolation techniques can also mitigate the impact of potential XSS attacks.

4.4.2 Defense: Authorization Context Consistency

Description Of Defense
One approach to mitigate privilege escalation attacks is to ensure that the web application *consistently* enforces its authorization policy. In this approach, the analysis does not focus on what the authorization policy is. Rather, it ensures that a policy is enforced consistently. This is particularly useful for situations where a policy exists, but it may be incorrectly enforced in parts of the web application because of developer error. A recent technique called MACE [27] checks for such consistencies by annotating authorization accesses of security sensitive actions within the source code. It then performs a control flow analysis to identify variables pertinent to determining authorization and constructs data dependency graph to capture data flows between authorization variables. By creating a source-sink graph form entry points (sources) to sensitive queries (sinks), it identifies authorization inconsistencies along each path from source to sink.

Description of Deployment Gaps
Such consistency analyses require access to the source code of the server (C.2, C.4). Consistency can only be checked for certain known operations (e.g. database DELETE, INSERT, etc.), so such analyses are limited in their generalizability for different types of privilege escalation (A.1).

Moreover, a single authorization check may result in numerous inconsistency validations propagated throughout the code, thereby artificially increasing the true positive rate (A.1). The annotation require manual hints from the developer, so it is not fully source compatible (C.1). Because the annotations have to be applied to *all* modules for the framework to work at all, it is not modular (C.5).

4.4.3 Defense: Web Server Isolation

Description Of Defense

Another approach to mitigate privilege escalation attacks and also possibly the impact of other types of web-based attacks is enforcing isolation in the web server. This trend has become popular in the past few years in the community. Similar to virtualization-based defenses that create isolation of machine in case of host-based attacks, web-based isolation techniques apply sandboxing among various web service components and limits the interactions to their bare minimum necessary. In other words, they apply the principle of least privileges in the context of web services. One of the pioneering work in this domain was the CLAMP system [29] which manually applies communication isolation, access control and sandboxing to the commonly used Linux, Apache, MySQL, and PHP (LAMP) system. A more recent work in this domain, Passe [4] automates most of this process by extracting data and control flow relationships in the web service and strongly enforcing such relationships. Passe enforces such isolations using process sandboxing, constraining database queries, and strong authentication and session management. Passe works with the Django web development framework. Isolations such as the one enforce by Passe can mitigate privilege escalation attacks. They can also mitigate the negative impacts of other web-based attacks such as XSS, code injection, logic vulnerabilities, session fixation, and file inclusion. Although in those contexts, isolation cannot stop the exploitation of the vulnerability, it can mitigate the damage done by the attack (e.g. the amount of data leaked, the parts of the database maliciously modified, the number of users compromised, etc.).

Description of Deployment Gaps

Techniques such as CLAMP and Passe have unknown effectiveness against web-based attacks other than privilege escalation (A.1). Even for privilege escalation, the scope of possible damage is unknown unless one can formally analyze the web server (A.1). More automated techniques like Passe can have high unacceptably high false positives (A.2). For example, if certain correct behavior is observed infrequently, Passe can block it as a potential attack. Both CLAMP and Passe require access to the server source code (C.2, C.4). Passe also requires developer-provided end-to-end test cases which are unscalable and hard to obtain (C.1).

4.4.4 Research Direction

Privilege escalation is one of the harder problems to solve as it relates to many other existing challenges such as proper input checks, logic vulnerabilities, missing authorization checks, etc.. Both mitigation techniques that address privilege escalation suffer from issues of effectiveness (A.1) as well as source (C.1), binary (C.2), and federation (C.4) compatability. Near-term research should focus on isolation techniques that are binary compatible and support federation. Even though such defenses will have unknown effectiveness against specific vectors of attack and they instead try to limit the damage afterward, they can be the basis for more quantifiable defenses in longer term.

4.5 LACK OF EFFECTIVE DEFENSES TO COMBAT LOGIC VULNERABILITIES

4.5.1 Attack Description

While logic flaws are somewhat loosely defined, they generally capture the set of vulnerabilities that violate business logic. They are often distinguished from other vulnerability types by the fact that other vulnerabilities are more broadly applicable to all users of a technology and are not closely tied to any particular logic. In the context of web security, logic vulnerabilities often result in unintended use of web applications without leveraging other attack types such as XSS or SQLi. For example, attackers have exploited vulnerabilities in web application logic to convince sites that payment should be directed to a malicious party rather than the site itself by exploiting flawed logic in the site's use of third-party payment APIs [45]. In this example, a vulnerable benign party fails to properly authenticate that they were paid by a web user, relying upon a third-party's claim (e.g. Amazon) that some payment was made, but failing to check to whom the payment was made (i.e. a malicious party rather than the benign site itself).

4.5.2 Defense: Logic Vulnerability Detectors

Description Of Defense
Much of the existing work for logic vulnerabilities involves trying to detect the vulnerabilities rather than implementing active protections. Most approaches rely upon developing a model of how the application should behave and testing for deviations in behavior. Approaches have been developed that create these models by analyzing source code [2, 14] and, in cases where source code is not available, by simply observing traffic [30]. For example, one approach [30] uses sample HTTP traffic to generate a website navigation graph, infers behavioral patterns, uses these patterns to generate test cases, and checks how a site performs against these tests using manually specified Linear Temporal Logic.

Description of Deployment Gaps
Model checkers can be useful for automatically finding logic vulnerabilities in web applications that contain formal specifications of expected behavior and state changes. Lacking such specifications, current logic vulnerability discovery approaches are forced to infer models of web applications [2,14] and to guess at expected behavior for narrow use cases [30]. This inference is prone to false negatives (A.3) and often requires substantial manual effort (C.1), both of which have hindered deployment. Model checkers would greatly benefit from approaches that enable the creation of formal specifications with a level of manual effort that is cost effective for small organizations, given the range of organizations that provide web applications on the Internet.

4.5.3 Defense: Third-Party Communication Defenses

Description Of Defense
One of the fundamental challenges of web security is that code is combined from multiple sources to create a complete web application. In order to minimize the amount of code that must be run within the same context, standards bodies, browsers, and web service providers have offered more

limited means for cross-origin communication (e.g. via the *postMessage* interface or web service APIs such as Google Checkout). However, even with these more limited means of communication, logic vulnerabillities remain in the way that parties from different origins communicate. Defenses have arisen that try to protect or verify the secure use of these interfaces. One recent example [47] looks at the use of single sign-on (e.g. Facebook) and payment services (e.g. Google Checkout) by third party websites and tries to identify invariants based upon sample network traces of sites interacting with these services; using these invariants it would build a model and monitor traffic patterns for deviations.

Description of Deployment Gaps
Existing defenses have not been able to provide much information about the false negatives and false positives rates. Given that existing approaches have been implemented for rather narrow use cases, one would expect that there would be significant false negatives. The approach described above [47] additionally requires a proxy server to be installed by websites (C.3, C.4).

More generally, the complex nature of developing interconnected web applications demands extensive security expertise from web developers, many of whom lack any background in security. Interfaces that are documented by standards bodies and implemented in browsers, web programming languages, and libraries often lack a means for using the interface securely by default. For example, the *postMessage* interface in HTML5 provides a controlled way for content from different origins to exchange messages with one another. To use *postMessage* securely developers should properly check that a message comes from the expected origin. While this practice is recommended by the WHATWG that is responsible for maintaining the HTML5 standard, it has not been well adopted by even the largest websites. Of the Alexa top 10,000 sites in 2013, more than half the receivers of *postMessage* communications were not checking or were improperly checking origins [38]. Given the large number of interfaces that can be misused in web applications and the limited security expertise of many developers, default means for securely using these interfaces are necessary for improving the security of deployed web applications.

4.5.4 Research Direction

Logic vulnerabilities are also one of the harder classes of attacks to address in short term. Since the vulnerability arises from the design of the system, and not the implementation, many automated techniques for detecting such vulnerabilities suffer from high false positive rates. Indeed, both defenses studied in this report suffer from high false negative rates (A.3). Future research should focus on observable that do not necessarily prove the existence of logic vulnerabilities, but can correlate well with their existence. For example, studying large web-based code bases may indicate that historically usage of certain functions or coding practices correlated with the number of logic vulnerabilities. Such analyses can then be used to assess the amount of logic vulnerabilities in the code. As a result, we recommend that in near-term, research is focused on statistical analyses of logic vulnerability occurrences.

4.6 LACK OF EFFECTIVE DEFENSES TO COMBAT FILE INCLUSION

4.6.1 Attack Description

File inclusion vulnerabilities [21] exist in web applications which open a file based on a user input without properly sanitizing it. For example, if a PHP scripts receives an input from the user (e.g. using `$_GET`) and then opens that file (e.g. using a `include`) without sanitizing it, an attacker can control what file is actually opened by including reference to another file. Consider the following example which allows a user to select his/her `ROLE` as either "analyst" or "auditor". Then it opens the appropriate .php file to show the appropriate content accordingly.

```
 <form method="get">
   <select name="ROLE">
      <option value="faculty">analyst</option>
      <option value="student">auditor</option>
   </select>
   <input type="submit">
</form>
```

```
 <?php
   if ( isset( $_GET['ROLE'] ) ) {
      include( $_GET['ROLE'] . '.php' );
   }
?>
```

The server can be forced to include the malicious external file by requesting the page: `/vulnerable.php?ROLE=http://malicious.com/exploit`. This is because the above PHP scripts opens the file included in `ROLE` without properly whether it is one of the correct values of "analyst" or "auditor".

Since the possible values for the file is often limited, developers can properly check the requests against those values if they are aware of the file inclusion vulnerabilities. The challenge, as in many other attacks, arises from securing many existing code bases and web servers.

We are not aware of any prominent defensive technique against file inclusion other than simple best practices. Best practices based on proper input checking before file opening would, of course, require access to the source code of the web server and are, therefore, not binary compatible (C.2). Moreover such countermeasures also do not support federation because they require control of the server (C.4). File inclusion attacks can also be stopped by regular expressions or input checks agnostic of the web server; however, such countermeasures have very high false positive rates (A.2).

4.6.2 Research Direction

File inclusion is a smaller problem compared to other web-based attacks because it only applies to situations where user input is directly used to open a file on the system. This can explain why no technqiues in the past few years have been proposed specifically against file inclusion attacks. We recommend that near term research focuses on studying the prevelence of such attacks/vulnerabilities before investing in new defenses.

4.7 LACK OF EFFECTIVE DEFENSES TO COMBAT DRIVE-BY DOWNLOADS

4.7.1 Attack Description

A *drive-by download* attack occurs when a user is lured into visiting a malicious website that serves code, typically JavaScript, that exploits an underlying vulnerability in the user's web browser or browser plugins. The vulnerability may be a traditional type of memory corruption, for example, such as a stack- or heap-based buffer overflow. If the exploit is successful, it often downloads additional malware onto the user's machine. Drive-by downloads may also be delivered through e-mail, but the key characteristic is that the user is tricked into visiting a malicious website that exploits vulnerable client-side software. Note, however, that drive-by downloads are not limited to specifically suspicious types of websites. They are, indeed, observed on various types of websites with different contents [26]. A variety of research efforts have been proposed to mitigate the harmful effects of visiting malicious web pages, the most significant of which are discussed below.

4.7.2 Defense: Static Analysis Based Detection

Description Of Defense

Zozzle [9] is a browser-based JavaScript attack detection tool that trains a machine learning classifier on features of the JavaScript abstract syntax tree (AST) using a corpus of labeled malware samples collected by the Nozzle heap-spraying detector [31]. Since sophisticated malware attempts to evade detection with obfuscation techniques, Zozzle is implemented within the browser's JavaScript runtime engine, which can dynamically analyze the JavaScript right before execution.

Rozzle [22] augments static analysis based detection techniques with multi-execution analysis. The main idea behind Rozzle is to explore all branches of every conditional that depends on the execution environment. For example, a JavaScript statement may be conditioned on the browser's user agent string. This multi-execution analysis allows for analysis of all potential code branches that may lead to malicious behavior.

Description of Deployment Gaps

Zozzle is evaluated on a corpus of malicious JavaScript samples and 1.2 million benign samples and the false positive rates for detection are very low (effectively zero). However, the false negative rate is 9%, which may be too high for practical deployment as many malicious samples are not detected (A.1, A.3). The prototype implementation is built within the Internet Explorer browser, which introduces compatibility challenges for users of other browsers (C.2) and cannot be applied without controlling the user's browser (C.4).

Rozzle offers acceptable performance overhead in terms of CPU and memory usage. However, Rozzle is built on top of the Chakra JavaScript engine in Internet Explorer 9, which makes the prototype incompatible with other browsers (C.2) and cannot be applied without controlling the user's browser (C.4).

4.7.3 Defense: Malicious Webpage Search

Description Of Defense
Another approach to mitigating drive-by downloads is to identify the malicious webpages through search engine queries. EvilSeed [18] starts with a set of known malicious we pages and then extracts the characteristic similarities of these pages and uses web crawling and existing search engines to locate additional, unknown malicious webpages that have the similar features.

Description of Deployment Gaps
One the main limitations of techniques that use web crawlers or search engines to automate the process of locating malicious webpages is that the webpage may have the ability to identify the crawler and present a different, non-malicious page. This limitation relates to the overall effectiveness of the technique and the potential for evasion (A.1).

4.7.4 Research Direction

Drive-by download attacks reside in the gray area between web-based and host-based attacks. Many such attacks use host-based exploitation techniques (e.g. memory corruption) to cause the execution of malicious code on the target system. Since the scope of such attacks can be very large, we recommend that in near term, research focuses on techniques to isolate the damage from such attacks to the browser environment and prevent its propagation to the rest of the system. Moreover, studies analyzing the prevalence of these attacks are more than seven years old. New measurement studies establishing their occurrences in the current environment can also be an informative near term research effort.

4.8 LACK OF EFFECTIVE DEFENSES TO COMBAT CLICKJACKING

4.8.1 Attack Description

Clickjacking attacks trick a user into clicking a link they did not intend to click or into using unintentional parameters in a request. The typical attack model involves a user visiting a malicious site that includes content from other, often trusted sites. For example, a malicious site might trick a user into unknowingly clicking a link that permits access to a webcam via an embedded Adobe Flash Player plugin settings page.

These attacks are accomplished by either changing the appearance of a page (e.g. displaying a fake cursor) or via careful timing manipulation (e.g. a malicious link appears right at the moment that a user is about to click). Similar variants exist for touchscreen devices and are commonly referred to as tapjacking attacks [34]. A 2012 study reported that 70% of the top 20 banking websites lacked protection against clickjacking attacks [48].

4.8.2 Defense: Anti-Framing Defenses

Description Of Defense
One of the most effective defenses to date allows websites to specify if its pages can be framed by other sites (cross-origin). A primary means for accomplishing clickjacking is for the malicious site to frame a page from a trusted site and then obfuscate that frame by tricks such as making it transparent or partially covering its values. A relatively successful anti-framing defense, X-Frame-Options, is a recently added HTTP header that enables benign websites to specify if their pages can be framed, thus preventing unintended framing by malicious sites.

Description of Deployment Gaps
The major gap in such defenses is that they are incompatible with sites that need to be framed (C). For example, social media sites wish to allow other sites to embed their *Like* or *Follow* pages as frames. Other gaps are that the X-Frame-Options must be specified per-page (C.1), requiring greater developer effort, which has hindered adoptability. Furthermore, proxy servers can undermine sites that properly implement X-Frame-Options by stripping out header fields (A.1). Alternative approaches for preventing framing, called framebusting techniques, typically rely upon JavaScript checks. The X-Frame-Options header typically provides better protection than these framebusting techniques as several ways to bypass these techniques have been demonstrated [33] (A.1).

4.8.3 Defense: User Confirmation Defenses

Description Of Defense
An existing defense that can be used by sites requiring embedded frames (and thus unable to deploy X-Frame-Options) is to prompt for user confirmation when links are clicked. For example, Facebook confirms if a user indeed wants to click the "Like" button for sites that it deems suspicious.

Description of Deployment Gaps
However, even with confirmations users can be tricked (A.1); one study reported that 47% of users

fell for a "double-clicking" attack [17]. Additionally, the need to prompt for confirmations has major usability implications and can only be deployed sparingly, preventing it from being adopted more widely (B.3).

4.8.4 Defense: Timing Delay Defenses

Description Of Defense
Recent defenses have suggested adding a delay until links become active following a visual change to the UI. The goal is to give the user enough time to register any timing-based obfuscations (e.g. a button that appeared under their cursor right as they were about to click). Firefox has implemented such a delay for installing add-ons and activating new features.

Description of Deployment Gaps
Other browsers have not followed suit, most likely due to the fact that these defenses can only be effective against timing-based attacks (A.1) and also have usability implications due to the forced delay.

4.8.5 Defense: Sensitive UI Area Defenses

Description Of Defense
Huang et al. [17] have proposed extending the timing concept by enforcing delays on links whenever the cursor enters a sensitive UI area. The sensitive area must be specified by the website developers. They have also suggested other defenses based upon this sensitive UI area concept that could defend against attacks that are not purely timing based. One is to have the browser compare screenshots of the actual displayed UI with an isolated rendering of that same embedded frame. The goal is to detect any visual obfuscations performed by another frame. Additionally, they recommend that any time the cursor enters the sensitive area that browsers force the cursor to be displayed and that the area be highlighted in an effort to better direct user attention to a potential sensitive object click.

Description of Deployment Gaps
Some of these recommendations are currently being adopted in the W3C UI Safety specification. However, gaps remain in the effectiveness of these proposals. Akhawe et al. [1] have argued that even with an accurately displayed frame and these timing protections that the user's perception can be manipulated by techniques such as showing additional pointers or distracting attention (A.1). Furthermore, the web application itself must be modified to properly delineate the sensitive UI area (C.1, C.4).

4.8.6 Research Direction

Since most reports assign low priority to clickjacking attacks, first research should focus on studying their prevalence in modern systems before building more effective defenses.

4.9 LACK OF EFFECTIVE DEFENSES TO COMBAT PLUG-IN ATTACKS

4.9.1 Attack Description

Plug-in attacks are a comparatively smaller class of web-based attacks targeting the clients in which a malicious plug-in is installed in the web browser [25]. Malicious plug-ins may be installed quietly or with the user consent. With more browsers blocking quite plug-in installs, malicious plug-in take the form of useful plug-in with some Trojan-like behavior. Malicious plug-in can have a wide range of unwanted behavior ranging from malicious actions such as stealing passwords, stealing email addresses, or uninstalling other extensions, to a potentially suspicious behavior of injecting dynamic JavaScripts or requesting non-existent domains.

Avoiding plug-ins and extensions altogether is a simple countermeasure against malicious plug-ins. However, with a wide range of useful functions provided by such plug-ins, avoiding them can negatively impact the usability of systems.

4.9.2 Defense: Eliciting Malicious Behavior

Description Of Defense
A defense against malicious plug-ins, is to elicit their malicious behavior by a technique similar to fuzzing [20]. This defense (called Hulk), leverages a concept called *honey pages* which dynamically adapts to a plug-in's expectation in order to trigger a potentially malicious behavior. Then by incorporating event handler fuzzing, Hulk provides the proper stimuli for the plug-in. Hulk detects potentially malicious behavior by monitoring the browser extension hooks, content scripts, and network activities. In many cases, it is unknown whether the behavior is actively malicious or buggy. For example, plug-ins that produce HTTP 4xx errors may just simply be buggy or they may be accessing a command and control server that does not exist yet, but will become available at a future date to implement a botnet-like behavior. Therefor, Hulk can mark a plug-in as malicious, benign, or suspicious, the last category referring to these uncertain situations.

Description of Deployment Gaps
Hulk requires access to the browser source code (C.1). It must also be applied to the entirety of the browser, so it is not modular (C.5). A larger problem is the potentially high false positive rate in a technique like Hulk (A.2). Since in many cases the real behavior of the analyzed plug-ins is unknown, it is hard to measure the real false positive rate. Also, it is uncertain what a user can do in cases where the plug-in looks suspicious, but not actively malicious. Finally, it is unknown how easy it is to evade Hulk-like detection mechanisms (A.1).

4.9.3 Defense: Micro-Privileges

Description Of Defense
Another defense against malicious browser extensions, implements fine-grained privileges to limit their accesses and potential damages [25]. The motivation behind this countermeasure is that Chrome's access control system fails to properly implement the principle of least privileges and privilege isolation. By allowing finer-grained privileges, this defense can mitigate different classes of

web-based attacks that use malicious extensions. For example, a user can limit the communications of an extension to certain domains, mitigating attacks such as XSS and CSRF in that extension.

Description of Deployment Gaps
The micro-privilege countermeasure has numerous usability and scalability problems. First, properly deciding what privileges an extension can have is left at the discretion of the users which has human overhead (B.3), and can severely impact security (A.1) and false negative rates (A.3) as the user does not have a proper basis for making such decisions. Moreover, such micro-privileges do not protect the system against malicious origins (i.e. if the allowed domain is itself malicious) (A.1). The countermeasure also requires access to the source code of the browser (C.2).

4.9.4 Research Direction

Since malicious plugins can perform a wide variety of malicious actions, research to address this gap should focus on both technological and policy aspects. For example, in managed environments such as DoD systems, plugins can be automatically vetted before being allowed installation on end clients. Some of the practicality challenges of existing defenses (e.g. source compatibility) may also be more tractable if it is only performed on a central vetting system.

5. SUMMARY AND CONCLUSION

Web-based attacks have been used widely in today's systems and network to compromise the security of web services. They can be used to exfilterate sensitive information, maliciously modify information, or damage the availability of important servers and services.

In this report, we studied some of the most prominent defenses proposed or deployed against web-based attacks. We particularly focused on deployment gaps and impracticalities in existing defenses that impede their widespread adoption.

Unfortunately, simple defenses against web-based attacks such as input sanitization, character escaping, and learning-based techniques provide little protection against evasion techniques and often have impractically high false positive and false negative rates.

More effective defenses in this domain such as taint-tracking, automated consistency checks, and static analysis techniques often either require significant modifications to servers and infrastructures, thus violating federation support, or they impose high computational or operator overheads.

Another class of defense against web-based attacks do not mitigate the vulnerability per se; rather, they try to limit the potential damages of an exploited vulnerability by creating fine-grained isolation, authorization, and access control among various components of a web service. Although beneficial from a security standpoint, these defenses fail to mitigate specific attacks. As a result, their precise impact and operational parameters are often unknown.

Despite many existing defenses and techniques, the domain of web-based attacks requires significant research and development effort to provide practical and effective defenses. Lack of effectiveness and ease of evasion of many existing defenses indicate a lack of deep understanding of this domain and of the challenges faced by organizations trying to mitigate real-world attacks. It is clear that stronger and more practical defenses are needed in all areas discussed in this report as none of the defenses reviewed can be expected to provide more than incremental protection in the near term. The complexity and distributed nature of the design and implementation of web-based systems exacerbate these challenges. Perfect security is an unattainable goal in general but is particularly elusive in the context of web security. Control over both the web server hosting the content as well as the web browsers accessing the content is typically not held by individual organizations and many attacks continue to be possible using vulnerabilities on either the server or browser end.

Given the poor state of security in the web-based systems and the rampant problems on every component of such systems, major improvements are hard to achieve without significant investment in this area. We recommend that in immediate term, efforts should focus on limiting a potential attacker's access to such systems using traditional air-gapping techniques and physical separation. In short and medium-term, research should focus on designing more effective defenses by considering the weaknesses in existing ones. Enumerating the previous "dead-ends" can facilitate such research as it can prevent duplication of effort based on weak defensive paradigms. Weaknesses identified in this report can guide such future efforts. Although strictly speaking such techniques can improve the state of security in web-based systems, we expect that given the extent of security problems, many such defenses may still be limited in their scope and a combination of them will be necessary for an effective coverage. In the long term, improved protocols, clean-slate design of standards, and systematic methodologies (e.g. formal methods) for web-based system design should be deployed to mitigate large classes of web-based attacks; similar to complete memory safety techniques against memory corruption attacks.

For high priority attacks, in near term, we recommend that research efforts focus on building more effective defenses against XSS and Session Fixation attacks. This can be achieved by focusing on fundamental properties of these attacks (spoofed identities) as opposed to some artifact of existing attacking techniques (e.g. special characters) that can be evaded by a motivated attacker. For code injection/reuse attacks, we recommend that research first focuses on studying and analyzing the scope and possibilities of such attacks. Although ad-hoc efforts have shown different possibilities for code injection/reuse attacks in various components of a web-based systems, the extent and potential impact of such attacks are much less understood in the web-based domain, compared to the host-based domain.

For medium priority attacks (privilege escalation, logic vulnerabilities, and file inclusion), we have a diverse set of recommendations in near term. Since privilege escalation can encompass many challenges, we recommend that compatible isolation techniques are developed that can be applied federally to different systems. Although the exact impact of such techniques will be unknown as they are not attempting to stop specific exploitation methods, we believe that they can form an important basis for tackling the problem in near term. For file inclusion attacks, since they are limited to specific types of web services, we recommend that their prevalence is evaluated before new techniques are designed. For logic vulnerabilities, since the problem is in the design of a system, not its implementation, we recommend that research focuses on finding observable properties that correlate with the existence of such vulnerabilities.

For low priority attacks, we recommend that research focuses on evaluating their prevalence in the wild before developing new defensive techniques. For malicious plugins, we recommend that proper policies (e.g. vetting before install, or white list) are deployed in conjunction with technological solutions to make the problem more tractable.

APPENDIX A: SUMMARY OF DEPLOYMENT GAPS

The table A.1 indicates a summary of the web-based countermeasures and their deployment gaps. The abbreviations in the first column stand for: Cross-Site Scripting (XSS), Code Injection (CI), Drive-by Downloads (DD), Click-jacking (CJ), Session Fixation (SF), File Inclusion (FI), Privilege Escalation (PE), and Plug-Ins (PI).

TABLE A.1

A summary of web-based countermeasures and their deployment gaps.

		Deployment Gaps										
		Protection			Overhead			Compatibility				
		A.1	A.2	A.3	B.1	B.2	B.3	C.1	C.2	C.3	C.4	C.5
XSS	XSS Auditor	×	×	×					×			
	Script Separation	×			×			×			×	
	Training-Based	×			×		×		×			
	Taint Tracking	×			×			×				×
	iFrame Isolation	×						×	×	×	×	
	Mutation-Based XSS				×		×		×		×	
CI	Static Analysis		×	×	×				×			
	Taint Tracking		×					×	×			
DD	Static Analysis	×		×					×		×	
	Webpage Search	×										
CJ	Anti-Framing	×						×				
	User Confirmation	×					×					
	Timing Delay	×										
	Sensitive UI Area	×						×			×	
Logic	Detectors			×				×				
	Third-Party Comm.									×	×	
SF	Code-Level							×	×		×	
	Reverse-Proxy				×					×		
FI	Generic		×						×		×	
PE	Context Consistency	×						×	×		×	×
	Isolation	×	×					×	×		×	
PI	Eliciting Behavior	×	×					×				×
	Micro-Privileges	×	×	×					×			

This page intentionally left blank.

REFERENCES

[1] Devdatta Akhawe, Warren He, Zhiwei Li, Reza Moazzezi, and Dawn Song. Clickjacking revisited a perceptual view of ui security. *BlackHat USA, August*, 2013.

[2] Davide Balzarotti, Marco Cova, Viktoria V Felmetsger, and Giovanni Vigna. Multi-module vulnerability analysis of web-based applications. In *Proceedings of the 14th ACM Conference on Computer & Communications Security*, pages 25–35. ACM, 2007.

[3] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 75–88, New York, NY, USA, 2008. ACM.

[4] Aaron Blankstein and Michael J Freedman. Automating isolation and least privilege in web services. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 133–148. IEEE, 2014.

[5] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 30–40, New York, NY, USA, 2011. ACM.

[6] Hristo Bojinov, Elie Bursztein, and Dan Boneh. Xcs: Cross channel scripting and its impact on web applications. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 420–431, New York, NY, USA, 2009. ACM.

[7] Yinzhi Cao, Vinod Yegneswaran, Phillip A Porras, and Yan Chen. Pathcutter: Severing the self-propagation path of xss javascript worms in social web networks. In *NDSS*, 2012.

[8] Cenzic. Application vulnerability trends report: 2014. Technical report, Cenzic, 2014.

[9] Charles Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Zozzle: Low-overhead mostly static JavaScript malware detection. In *Proceedings of the Usenix Security Symposium*, August 2011.

[10] Johannes Dahse and Thorsten Holz. Static Detection of Second-Order Vulnerabilities in Web Applications. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 989–1003, San Diego, CA, August 2014. USENIX Association.

[11] Johannes Dahse, Nikolai Krein, and Thorsten Holz. Code reuse attacks in php: Automated pop chain generation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 42–53, New York, NY, USA, 2014. ACM.

[12] Adam Doupé, Weidong Cui, Mariusz H. Jakubowski, Marcus Peinado, Christopher Kruegel, and Giovanni Vigna. dedacota: Toward preventing server-side xss via automatic code and data separation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 1205–1216, New York, NY, USA, 2013. ACM.

[13] S Esser. Utilizing Code Reuse or Return Oriented Programming in PHP Applications. BlackHat USA, 2010.

[14] Viktoria Felmetsger, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Toward automated detection of logic vulnerabilities in web applications. In *USENIX Security Symposium*, pages 143–160, 2010.

[15] WG Halfond, Jeremy Viegas, and Alessandro Orso. A classification of sql-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, pages 65–81. IEEE, 2006.

[16] Mario Heiderich, Jörg Schwenk, Tilman Frosch, Jonas Magazinius, and Edward Z. Yang. mxss attacks: Attacking well-secured web-applications by using innerhtml mutations. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 777–788, New York, NY, USA, 2013. ACM.

[17] Lin-Shung Huang, Alexander Moshchuk, Helen J Wang, Stuart Schecter, and Collin Jackson. Clickjacking: Attacks and defenses. In *USENIX Security Symposium*, pages 413–428, 2012.

[18] Luca Invernizzi, Stefano Benvenuti, Marco Cova, Paolo Milani Comparetti, Christopher Kruegel, and Giovanni Vigna. Evilseed: A guided approach to finding malicious web pages. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 428–442, Washington, DC, USA, 2012. IEEE Computer Society.

[19] Martin Johns, Bastian Braun, Michael Schrank, and Joachim Posegga. Reliable protection against session fixation attacks. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 1531–1537. ACM, 2011.

[20] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, Vern Paxson, Dhilung Kirat, Giancarlo De Maio, Yan Shoshitaishvili, Gianluca Stringhini, et al. Hulk: eliciting malicious behavior in browser extensions. In *Proceedings of the 23rd USENIX Security Symposium*, pages 641–654, 2014.

[21] Or Katz. Detecting remote file inclusion attacks. *White Paper. Breach Security Inc., May*, 2009.

[22] Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Rozzle: Decloaking internet malware. In *IEEE Symposium on Security and Privacy*, May 2012.

[23] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: Large-scale detection of dom-based xss. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 1193–1204, New York, NY, USA, 2013. ACM.

[24] Sebastian Lekies, Ben Stock, and Martin Johns. A tale of the weaknesses of current client-side xss filtering. In *Proceedings of BlackHat USA 2014*, 2014.

[25] Lei Liu, Xinwen Zhang, Guanhua Yan, and Songqing Chen. Chrome extensions: Threat analysis and countermeasures. In *NDSS*, 2012.

[26] Niels Provos Panayiotis Mavrommatis and Moheeb Abu Rajab Fabian Monrose. All your iframes point to us. In *USENIX Security Symposium*, pages 1–16, 2008.

[27] Maliheh Monshizadeh, Prasad Naldurg, and VN Venkatakrishnan. Mace: Detecting privilege escalation vulnerabilities in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer & Communications Security*, pages 690–701. ACM, 2014.

[28] Top OWASP. Top 10–2010–the ten most critical web application security risks. *The Open Web Application Security Project*, 2010.

[29] Bryan Parno, Jonathan M McCune, Dan Wendlandt, David G Andersen, and Adrian Perrig. Clamp: Practical prevention of large-scale data leaks. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 154–169. IEEE, 2009.

[30] Giancarlo Pellegrino and Davide Balzarotti. Toward black-box detection of logic flaws in web applications. In *Network and Distributed System Security (NDSS) Symposium*, 2014.

[31] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 169–186, Berkeley, CA, USA, 2009. USENIX Association.

[32] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, March 2012.

[33] Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. *IEEE Oakland Web*, 2:6, 2010.

[34] Gustav Rydstedt, Baptiste Gourdin, Elie Bursztein, and Dan Boneh. Framing attacks on smart phones and dumb routers: tap-jacking and geo-localization attacks. In *Proceedings of the 4th USENIX conference on Offensive technologies*, pages 1–8. USENIX Association, 2010.

[35] Prateek Saxena, David Molnar, and Benjamin Livshits. Scriptgard: Automatic context-sensitive sanitization for large-scale legacy web applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 601–614, New York, NY, USA, 2011. ACM.

[36] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[37] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. Diglossia: detecting code injection attacks with precision and efficiency. In *Proceedings of the 2013 ACM SIGSAC conference on Computer &#38; communications security*, CCS '13, pages 1181–1192, New York, NY, USA, 2013. ACM.

[38] Sooel Son and Vitaly Shmatikov. The postman always rings twice: Attacking and defending postmessage in html5 websites. In *NDSS*, 2013.

[39] Trustwave SpiderLabs. The web hacking incident database. semiannual report. july to december 2010. Technical report, Technical report, Computer Science in Trustwave SpiderLabs, 2011.

[40] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. Precise client-side protection against dom-based cross-site scripting. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 655–670, San Diego, CA, August 2014. USENIX Association.

[41] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 48–62. IEEE, 2013.

[42] Yusuke Takamatsu, Yuji Kosuga, and Kenji Kono. Automated detection of session fixation vulnerabilities. In *Proceedings of the 19th international conference on World wide web*, pages 1191–1192. ACM, 2010.

[43] Mike Ter Louw and VN Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing browsers. In *IEEE Symposium on Security and Privacy*, pages 331–346, 2009.

[44] W3C. Web Services Glossary. http://www.w3.org/TR/ws-gloss/, 2014.

[45] Rui Wang, Shuo Chen, XiaoFeng Wang, and Shaz Qadeer. How to shop for free online–security analysis of cashier-as-a-service based web stores. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 465–480. IEEE, 2011.

[46] Michael Weissbacher, Tobias Lauinger, and William Robertson. Why is csp failing? trends and challenges in csp adoption. In Angelos Stavrou, Herbert Bos, and Georgios Portokalidis, editors, *Research in Attacks, Intrusions and Defenses*, volume 8688, pages 212–233. Springer International Publishing, 2014.

[47] Luyi Xing, Yangyi Chen, XiaoFeng Wang, and Shuo Chen. Integuard: Toward automatic protection of third-party web service integrations. In *NDSS*, 2013.

[48] Dingjie Yang. Clickjacking: An Overlooked Web Security Hole. https://community.qualys.com/blogs/securitylabs/2012/11/29/clickjacking-an-over looked-web-security-hole, 2012.

This page intentionally left blank.